

# **Solving Partial Differential Equations Using The Chombo Framework for Block-Structured Adaptive Mesh Refinement Algorithms**

**Dan Martin, Brian van Straalen  
Applied Numerical Algorithms Group (ANAG)  
Lawrence Berkeley National Laboratory  
June 29, 2007**

available at

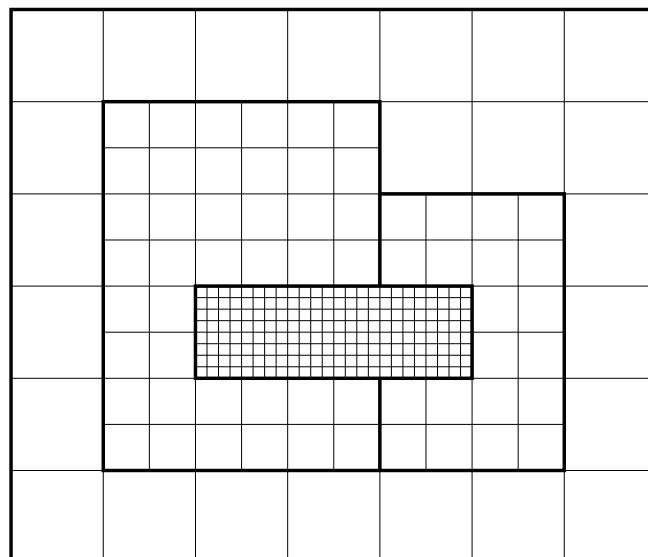
<http://seesar.lbl.gov/ANAG/staff/martin/talks/ChomboTutorial07.pdf>

# Adaptive Mesh Refinement (AMR)

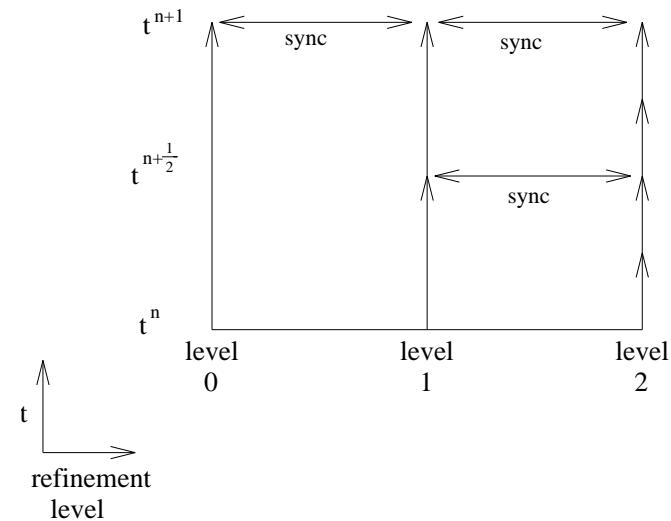
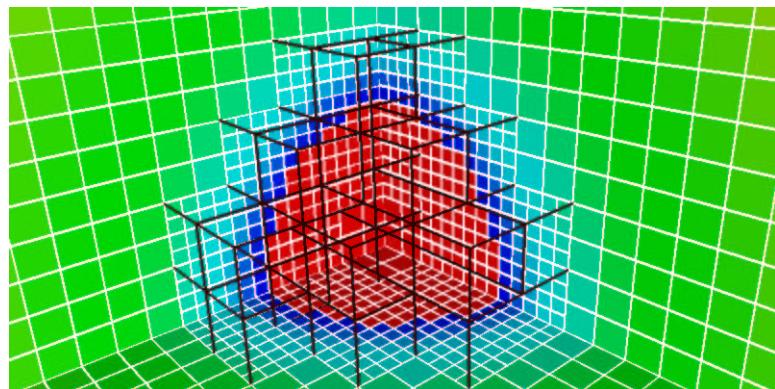
## (Berger & Oliger, 1984):

### Approach:

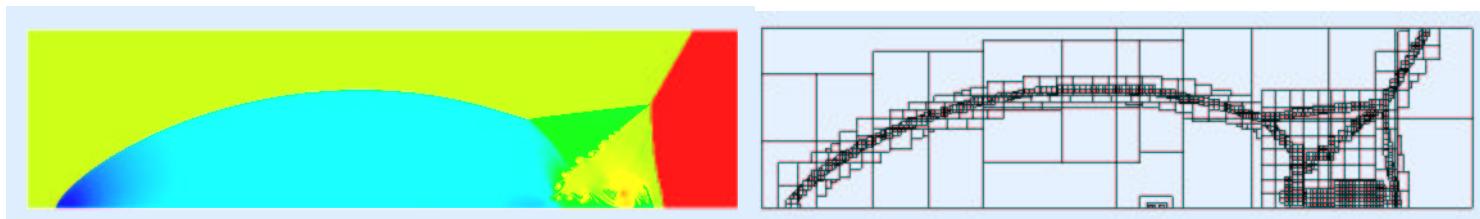
- locally refine patches of the domain where needed to improve solution
- each patch is a logically rectangular structured grid
  - better efficiency of data access
  - can amortize overhead of irregular operations over large number of regular operations
- refined grids are dynamically created and destroyed



## Block-Structured Local Refinement (Berger and Oliger, 1984)



Refined regions are organized into logically rectangular patches  
Refinement performed in time as well as in space.



## **Chombo: a Software Framework for Block-Structured AMR**

Requirement: to support a wide variety of applications that use block-structured AMR using a common software framework.

- Mixed-language model: C++ for higher-level data structures, Fortran for regular single-grid calculations.
- Reuseable components. Component design based on mapping of mathematical abstractions to classes.
- Build on public-domain standards: MPI, HDF5, VTK.
- Interoperability with other SciDAC ISIC tools: grid generation (TSTT), solvers (TOPS), performance analysis tools (PERC).

Previous work: BoxLib (LBNL/CCSE), KeLP (Baden, et. al., UCSD), FIDIL (Hilfinger and Colella).

## **Logistics/FAQ:**

- Chombo – Swahili word meaning “box”, “container”, or “useful thing”
- Available from <http://seesar.lbl.gov/ANAG/software.html>
- Freely available, subject to export controls and BSD-like license agreement
- Requirements: C++, Fortran compilers, PERL, HDF5 (for I/O), MPI
- Supports different precisions (float, double) through use of `Real` data type.
- Online doxygen documentation:  
<http://davis.lbl.gov/Manuals/CHOMBO-CVS>
- E-mail support: [chombo@davis.lbl.gov](mailto:chombo@davis.lbl.gov)
- Chombo users e-mail group: [chombousers@davis.lbl.gov](mailto:chombousers@davis.lbl.gov)
- My e-mail: [DFMartin@lbl.gov](mailto:DFMartin@lbl.gov)

## Layered Design

- **Layer 1.** Data and operations on unions of boxes – set calculus, rectangular array library (with interface to Fortran), data on unions of rectangles, with SPMD parallelism implemented by distributing boxes over processors.
- **Layer 2.** Tools for managing interactions between different levels of refinement in an AMR calculation – interpolation, averaging operators, coarse-fine boundary conditions.
- **Layer 3.** Solver libraries – AMR-multigrid solvers, Berger-Oliger time-stepping.
- **Layer 4.** Complete parallel applications.
- **Utility layer.** Support, interoperability libraries – API for HDF5 I/O, visualization package implemented on top of VTK, C API's.

## Chombo organization

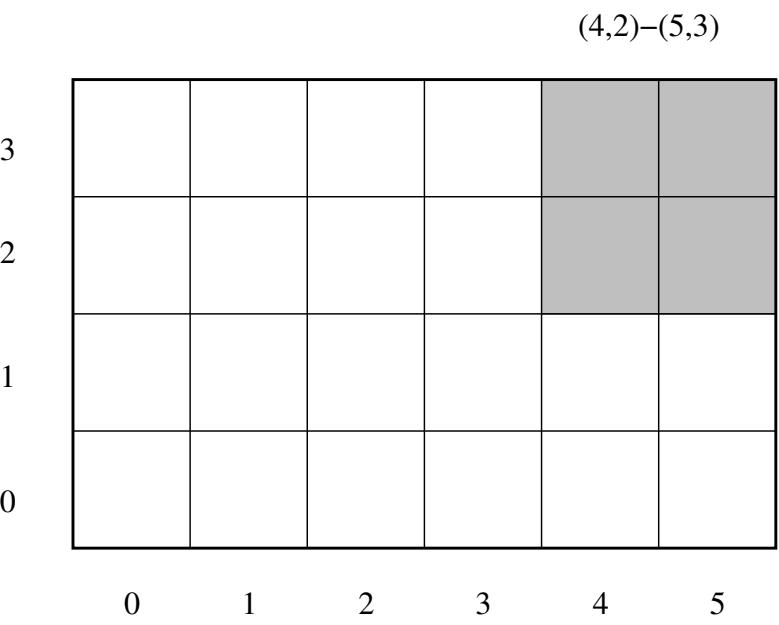
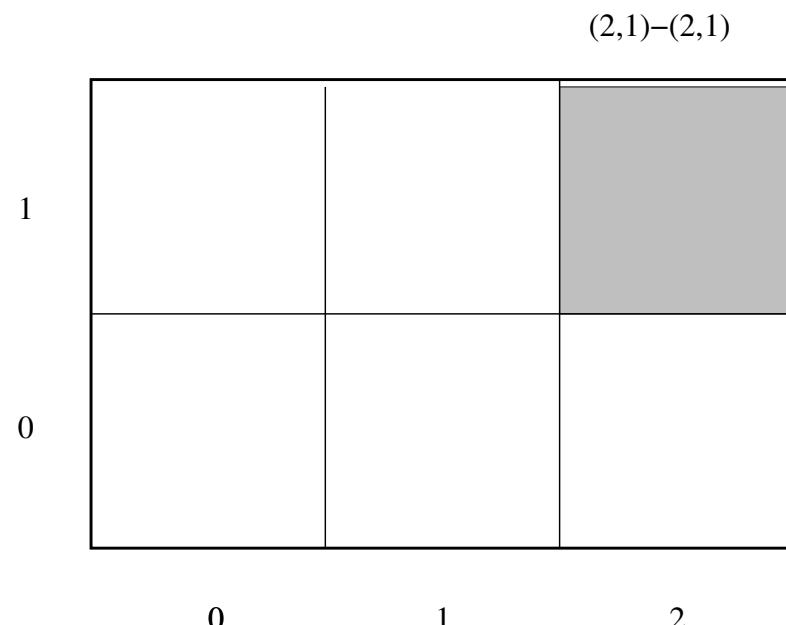
```
Chombo/          (root directory)
    lib/
        mk/
        src/
            BoxTools
            AMRTools
            EllipticDesign
            AMRTimeDependent

    example/      (various examples)
        AMRPoisson/
            designExec/
        PPMAMRGodunov/
            execPolytropic
        AMRUpwind/
            exec
```

## Layer 1 Classes (BoxTools)

Global index spaces:

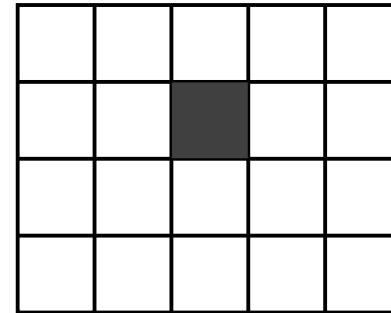
- Each AMR level uses a global index space to locate points in space
- Each level's index space is related to the others by simple refinement and coarsening operations.
- makes it easy to organize interlevel operations and to perform operations on levels which are unions of patches,



## IntVect class

Location in index space:  $i \in \mathbb{Z}^d$ .

Can translate  $i_1 \pm i_2$ , coarsen  $\frac{i}{s}$ , refine  $i * s$ .



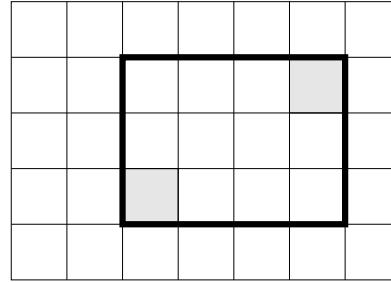
## 2D example:

```
IntVect iv(2,3);      \\ create IntVect
iv *= 2;              \\ multiply by a factor of 2 (now (4,6)
iv.coarsen(2);        \\ coarsen by factor of 2 (now 2,3)

IntVect iv2(1,2);    \\ second IntVect
iv += iv2;            \\ add iv2 to iv -- iv = (3,5)

int i = iv[0];        \\ access 0th component (i = 3)
```

## Box class



$B \subset \mathbb{Z}^d$  is a rectangle in space:  $B = [\mathbf{i}_{low}, \mathbf{i}_{high}]$ .

$B$  can be translated, coarsened, refined. Supports different centerings (node-centered vs. cell-centered) in each coordinate direction.

### 2D example:

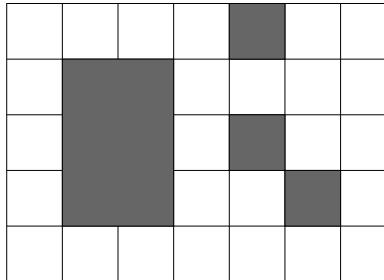
```
IntVect lo(1,1), hi(2,3); \\ IntVects to define box extents
Box b(lo,hi);           \\ define cell-centered box
b.refine(2);             \\ refine by factor of 2: now (2,2)-(3,5)
b.coarsen(2);            \\ coarsen: now back to (1,1)-(2,3)
b.surroundingNodes()    \\ convert to node-centering -- (1,1)-(3,4)
b.enclosedCells()        \\ back to cell-centering -- (1,1)-(2,3)

Box b2(b)                \\ copy constructor
b2.shift(IntVect::Unit); \\ shift b2 by (1,1) -- now (2,2)-(3,4)
b.intersect(b2);          \\ intersect b with b2 -- b now (2,2)-(2,3)
```

`ProblemDomain` **class:**

- Class which encapsulates the computational domain.
- basic implementation: essentially a `Box` with periodicity information.
- In most cases, periodicity is hidden from the user and is handled as a wrapping of the index space.

`IntVectSet` Class:



$\mathcal{I} \subset \mathbb{Z}^d$  is an arbitrary subset of  $\mathbb{Z}^d$ .  $\mathcal{I}$  can be shifted, coarsened, refined. One can take unions and intersections, with other `IntVectSets` and with Boxes, and iterate over an `IntVectSet`. Useful for representing irregular sets.

**example:**

```
IntVect iv1, iv2, iv3; \\ various IntVsects  
Box b; \\ box region  
IntVectSet ivSet(b) \\ set of all Index locations in b  
ivSet |= iv1; \\ union operator  
ivSet.refine(2); \\ refinement of IntVectSet  
ivSet.coarsen(2); \\ coarsen back to original  
ivSet -= iv2; \\ remove iv2 from intVectSet  
ivSet.grow(1); \\ grow intVectSet by a radius of 1
```

## **BaseFab<T> container class:**

Templated multidimensional array container class for T (int, Real, etc) over region defined by a Box.

### **example:**

```
Box domain(IntVect::Zero, 3*IntVect::Unit); // box from (0,0)-(3,3)
int nComp = 2;                                // 2 components
BaseFab<int> intfab(domain, nComp);          // define container for int's
intfab.setVal(1);                             // set values to 1

BoxIterator bit(domain); // iterator over domain
for (bit.begin(); bit.ok(); ++bit)
{
    iv = bit();
    ival(iv,0) = iv[0];                      // set 0th component at iv to index
}
int* ptr = ifab.dataPtr(); // pointer to the contiguous block of
                           // data which can be passed to Fortran.
```

### FArrayBox class:

Specialized BaseFab<Real> with additional floating-point operations – can add, multiply, divide, compute norms

### example:

```
Box domain;  
FArrayBox afab(domain, 1); // define single-component FAB's  
FArrayBox bfab(domain, 1);  
  
afab.setVal(1.0); // afab = 1 everywhere  
bfab.setVal(2.0); // set bfab to be 2  
  
afab.mult(2.0); // multiply all values in afab by 2  
afab.plus(bfab); // add bfab to afab (modifying afab)
```

## Example: explicit heat equation solver on a single grid

// C++ code:

```
Box domain(IntVect::Zero, (nx-1)*IntVect::Unit);
FArrayBox soln(grow(domain,1), 1);
soln.setVal(1.0);

for (int nstep = 0;nstep < 100; nstep++)
{
    heatsub2d_(soln.dataPtr(0),
               &(soln.loVect()[0]), &(soln.hiVect()[0]),
               &(soln.loVect()[1]), &(soln.hiVect()[1]),
               domain.loVect(), domain.hiVect(),
               &dt, &dx, &nu);
}
```

```
c Fortran code:
```

```
    subroutine heatsub2d(phi,nlphi0, nhphi0,nlphil, nhphil,
&      nlreg, nhreg, dt, dx, nu)
```

```
    real*8  phi(nlphi0:nhphi0,nlphil:nhphil)
    real*8 dt,dx,nu
    integer nlreg(2),nhreg(2)
```

```
c Remaining declarations, setting of boundary conditions goes here.
```

```
    do j = nlreg(2), nhreg(2)
        do i = nlreg(1), nhreg(1)
            lapphi = (phi(i+1,j) +phi(i,j+1) +phi(i-1,j) +phi(i,j-1)
&                      -4.0d0*phi(i,j))/(dx*dx)

            phi(i,j) = phi(i,j) + nu*dt*lapphi
        enddo
    enddo

    return
end
```

## **ChomboFortran**

ChomboFortran is a set of macros used primarily by Chombo for:

- Managing the C++/Fortran interface
- Writing Dimension-independent (Fortran) code

Advantages to ChomboFortran:

- enables fast (2d) prototyping, and simple extension to 3d.
- Simplifies code maintenance and duplication by reducing the need for dimension-specific code

## **ChomboFortran C++/Fortran interface** (Previous example)

### **C++ side:**

```
heatsub2d_(soln.dataPtr(0),  
           &(soln.loVect()[0]), &(soln.hiVect()[0]),  
           &(soln.loVect()[1]), &(soln.hiVect()[1]),  
           domain.loVect(), domain.hiVect(),  
           &dt, &dx, &nu);
```

### **Fortran side** (heat.f):

```
subroutine heatsub2d(phi,iphilo0, iphihi0,iphilo1, iphihi1,  
&                      domboxlo, domboxhi, dt, dx, nu)  
  
real*8 phi(iphilo0:iphihi0,iphilo1:iphihi1)  
real*8 dt,dx,nu  
integer domboxlo(2),domboxhi(2)
```

Managing the interface is error-prone and dimensionally dependent (since 3d will have more index extents for array sizing)

With ChomboFortran Macros:

**C++ side:**

```
FORT_HEATSUB( CHF_FRA(soln) ,  
               CHF_BOX(domain) ,  
               CHF_REAL(dt) , CHF_REAL(dx) , CHF_REAL(nu) );
```

**ChomboFortran side (heatF.ChF):**

```
subroutine heatsub(CHF_FRA[phi] , CHF_BOX[domain] ,  
&                      CHF_REAL[dt] , CHF_REAL[dx] , CHF_REAL[nu] )
```

(Note that ChomboFortran declares the entire argument list as well)

# Dimension Independence with ChomboFortran:

- Looping macros: CHF\_MULTIDO
  - Data Access: CHF\_IX

## Replace:

```

do j = nlreg(2), nhreg(2)
    do i = nlreg(1), nhreg(1)
        phi(i,j) = phi(i,j) + nu*dt*lphi(i,j)
    enddo
enddo

```

**With:**

```

CHF_MULTIDO[dombox; i;j;k]
    phi(CHF_IX[i;j;k]) = phi(CHF_IX[i;j;k])
&                                         + nu*dt*lphi(CHF_IX[i;j;k])
CHF ENDDO

```

## **More ChomboFortran support for Dimension-independence:**

CHF\_DDECL and CHF\_DTERM Macros:

**Replace:**

```
integer i,j,k  
real*8 x,y,z
```

```
x = i*dx  
y = j*dx  
z = k*dx
```

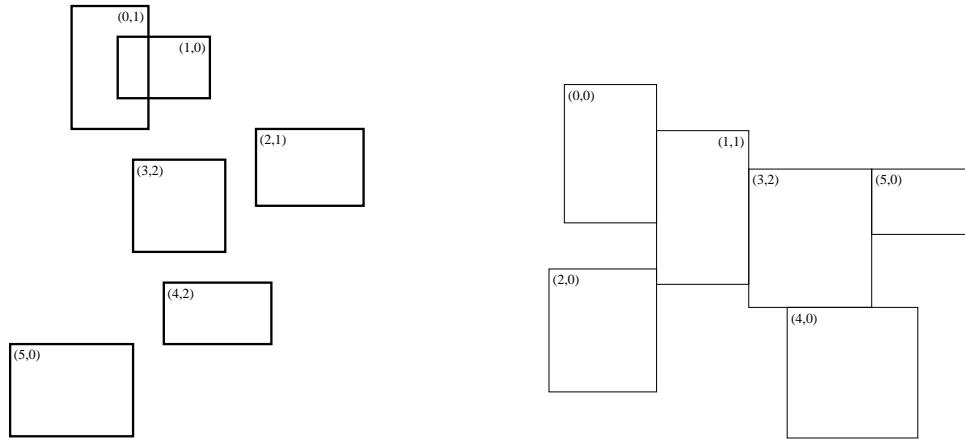
**With:**

```
integer CHF_DDECL[i;j;k]  
REAL_T CHF_DDECL[x;y;z]
```

```
CHF_DTERM[  
          x = i*dx;  
          y = j*dx;  
          z = k*dx ]
```

## Distributed Data on Unions of Rectangles

Provides a general mechanism for distributing data defined on unions of rectangles onto processors, and communications between processors.



- Metadata of which all processors have a copy: BoxLayout is a collection of Boxes and processor assignments:  $\{B_k, p_k\}_{k=1}^{nGrids}$ . DisjointBoxLayout : public BoxLayout is a BoxLayout for which the Boxes must be disjoint.
- template <class T> LevelData<T> and other container classes hold data distributed over multiple processors. For each  $k = 1 \dots nGrids$ , an "array" of type T corresponding to the box  $B_k$  is allocated on processor  $p_k$ . Straightforward API's for copying, exchanging ghost cell data, iterating over the arrays on your processor in a SPMD manner.

## **Software Reuse by Templating Dataholders**

Classes can be parameterized by types, using the class template language feature in C++.

`BaseFAB<T>` is a multidimensional array for any type `T`.

```
FArrayBox: public BaseFAB<Real>
```

In `LevelData<T>`, `T` can be any type that "looks like" a multidimensional array.

Examples include:

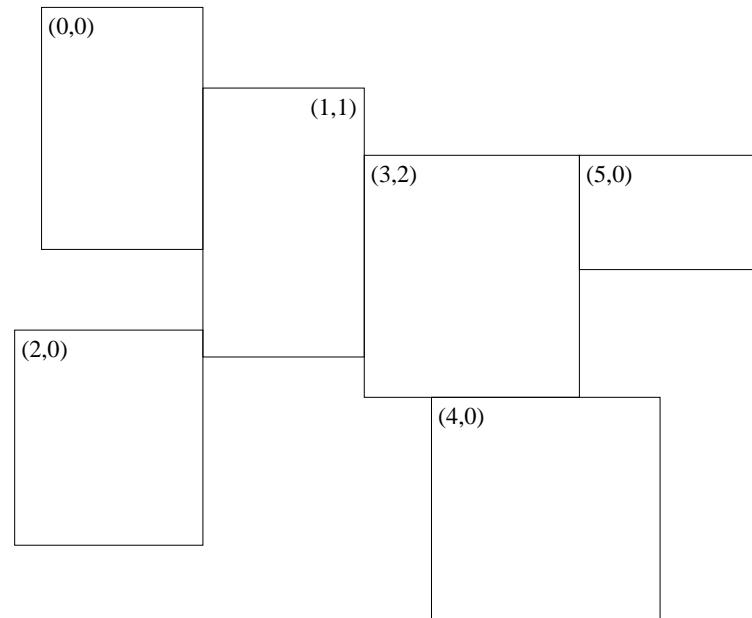
- Ordinary multidimensional arrays, e.g. `LevelData<FArrayBox>`.
- A composite array type for supporting embedded boundary computations
- Binsorted lists of particles, e.g. `BaseFab<List<ParticleType>>`

## `DisjointBoxLayout` class:

Set of disjoint Boxes which comprise the valid regions on a single level, including processor assignments for each box/grid/patch

2 ways to iterate through a `DisjointBoxLayout` –

- `LayoutIterator` – iterates through **all** boxes in the `DisjointBoxLayout`, regardless of which processor they are assigned to (useful for computing intersections, etc)
- `DataIterator` – iterates through only boxes on **this** processor (used when accessing data)



**example:**

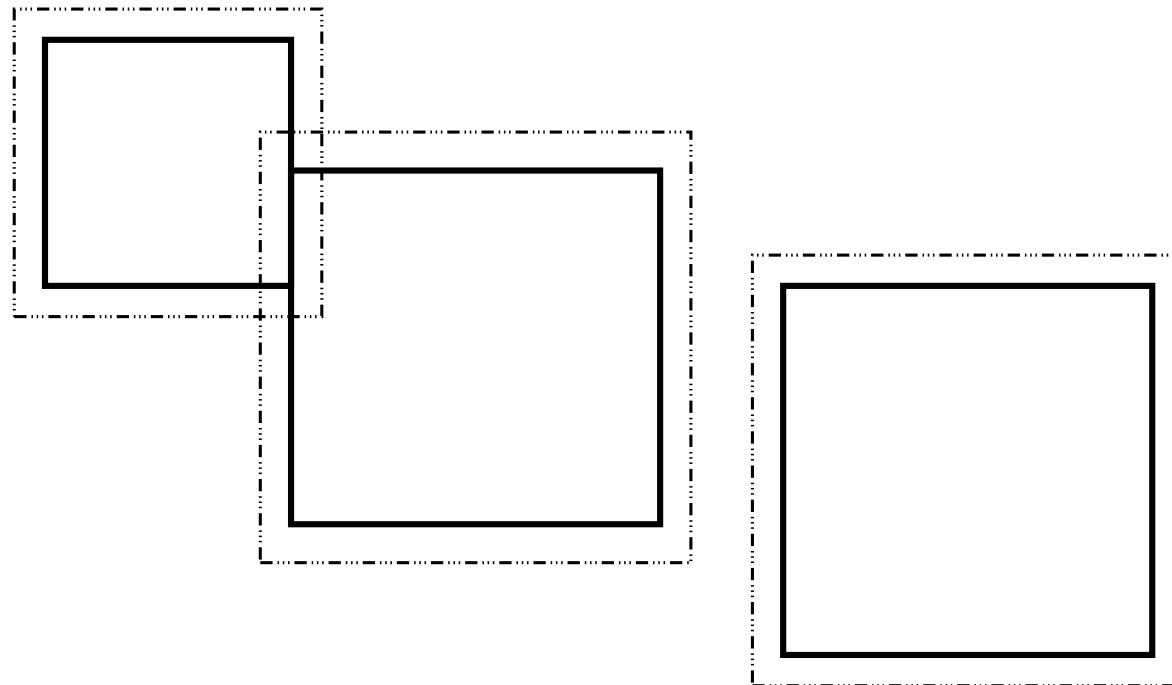
```
Vector<Box> boxes;           // boxes and processor assignments
Vector<int> procAssign;
ProblemDomain domain;
DisjointBoxLayout dbl(boxes, procAssign, domain); // define dbl

// access _all_ boxes
LayoutIterator lit = dbl.layoutIterator();
for (lit.begin(); lit.ok(); ++lit)
{
    const Box thisBox = dbl[lit];
}

// access only local boxes
DataIterator dit = dbl.dataIterator();
for (dit.begin(); dit.ok(); ++dit)
{
    const Box thisLocalBox = dbl[dit];
}
```

**LevelData<T> class:**

Distributed data associated with a `DisjointBoxLayout`. Can have ghost cells around each box to handle intra-level, inter-level, and domain boundary conditions.



**example:**

```
DisjointBoxLayout grids;
int nComp = 2; \\ two data components
IntVect ghostVect(IntVect::Unit) \\ one layer of ghost cells

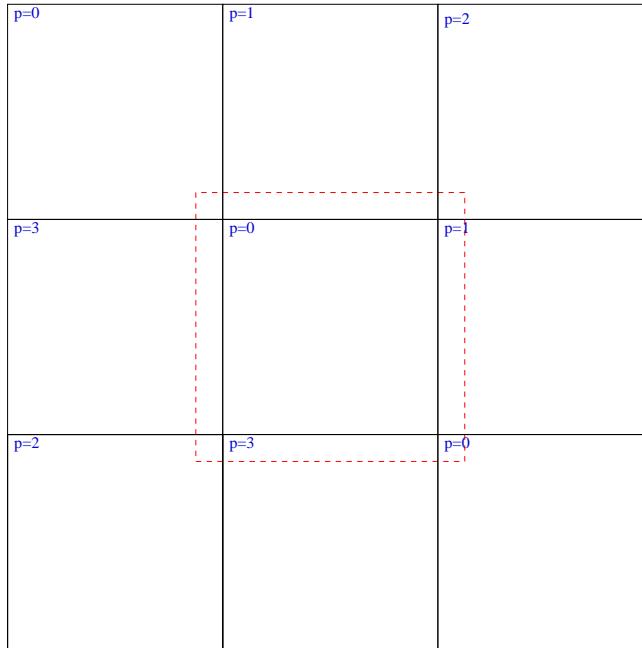
LevelData<FArrayBox> ldf(grids, nComp, ghostVect); // Real distribute
LevelData<FArrayBox> ldf2(grids, nComp, ghostVect);

DataIterator dit = grids.dataIterator(); // iterate local data
for (dit.begin(); dit.ok(); ++dit)
{
    FArrayBox& thisFAB = ldf[dit];
    thisFAB.setVal(procID());
}

// fill ghost cells with "valid" data from neighboring grids
ldf.exchange();

ldf.copyTo(ldf2); \\ copy from ldf->ldf2
```

## Example: explicit heat equation solver, parallel case



Want to apply the same algorithm as before, except that the data for the domain is decomposed into pieces and distributed to processors.

- `LevelData<T>::exchange( )`: obtains ghost cell data from valid regions on other patches.
- `DataIterator`: iterates over only the patches that are owned on the current processor.

```
// C++ code:  
Box domain(IntVect::Zero, (nx-1)*IntVect::Unit);  
DisjointBoxLayout dbl;  
// Break domain into blocks, and construct the DisjointBoxLayout.  
makeGrids(domain,dbl,nx);  
  
LevelData<FArrayBox> phi(dbl, 1, IntVect::TheUnitVector());  
  
for (int nstep = 0;nstep < 100;nstep++)  
{  
...  
// Apply one time step of explicit heat solver: fill ghost cell va  
// and apply the operator to data on each of the Boxes owned by th  
// processor.  
  
phi.exchange();
```

```
// Iterator iterates only over those boxes that are on
// this processor.

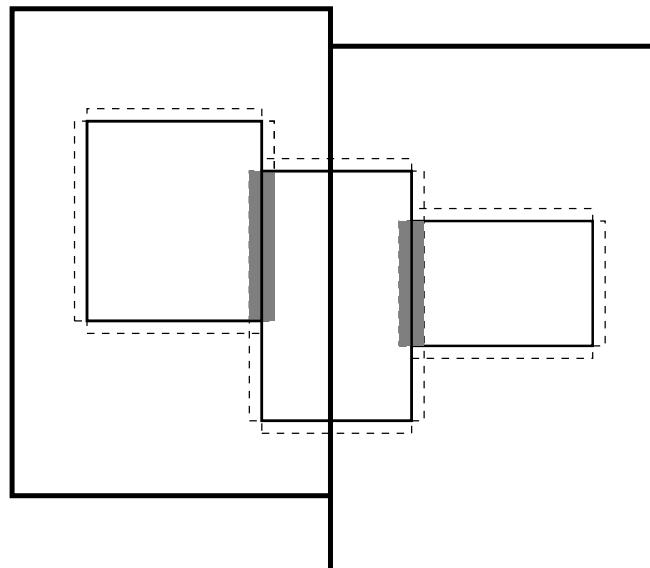
DataIterator dit = dbl.dataIterator();
for (dit.reset();dit.ok();++dit)
{
    FArrayBox& soln = phi[dit()];
    Box& region = dbl[dit()];
    FORT_HEATSUB(CHF_FRA(soln),
                  CHF_BOX(region),
                  CHF_BOX(domain),
                  CHF_REAL(dt), CHF_REAL(dx), CHF_REAL(nu));
}
}
```

## Layer 2: Coarse-Fine Interactions (AMRTools).

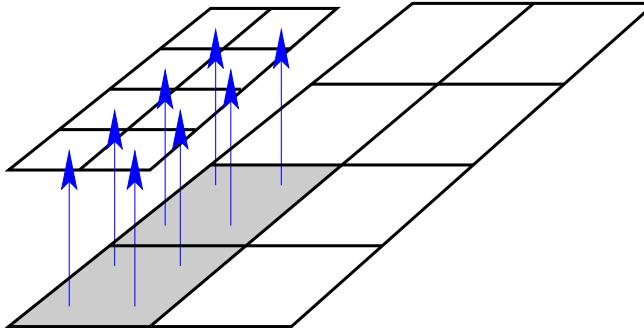
The operations that couple different levels of refinement are among the most difficult to implement AMR.

- Interpolating between levels (`FineInterp`).
- Averaging down onto coarser grids (`CoarseAverage`).
- Interpolation of boundary conditions (`PWLFillpatch`, `QuadCFInterp`).
- Managing conservation at coarse-fine boundaries (`LevelFluxRegister`).

These operations typically involve interprocessor communication and irregular computation.



**FineInterp class:**

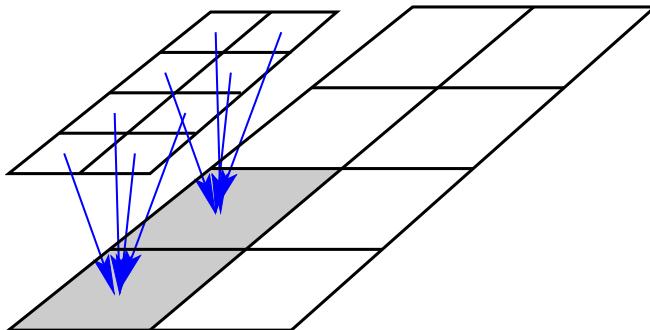


- Linearly interpolates data from coarse cells to the overlaying fine cells
- Useful when initializing newly refined regions after regridding

**example:**

```
ProblemDomain fineDomain;  
DisjointBoxLayout coarseGrids, fineGrids;  
int refinementRatio, nComp;  
LevelData<FArrayBox> coarseData(coarseGrids, nComp);  
LevelData<FArrayBox> fineData(fineGrids, nComp);  
  
FineInterp interpolator(fineGrids, nComp, refinementRatio,  
                        fineDomain);  
  
// fineData is filled with linearly interpolated coarseData  
interpolator.interpToFine(fineData, coarseData);
```

CoarseAverage **class**:



- Averages data from finer levels to covered regions in the next coarser level.
- Useful for bringing coarse levels into synch with refined levels

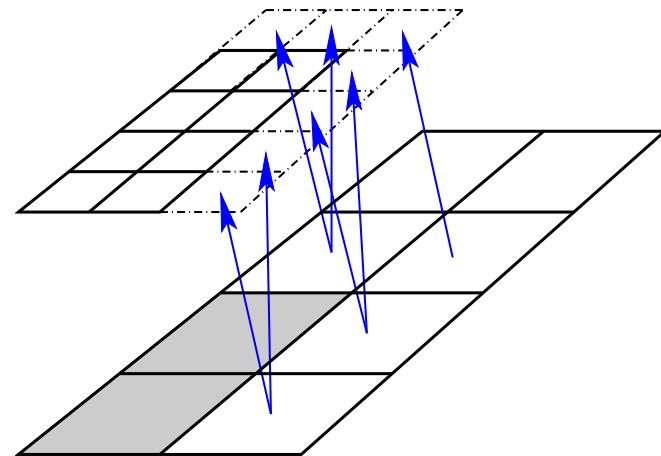
**example:**

```
DisjointBoxLayout fineGrids;  
DisjointBoxLayout crseGrids;  
int nComp, refRatio;
```

```
LevelData<FArrayBox> fineData(fineGrids, nComp);  
LevelData<FArrayBox> crseData(crseGrids, nComp);
```

```
CoarseAverage averager(fineGrids, crseGrids, nComp, refRatio);  
  
averager.averageToCoarse(crseData, fineData);
```

**PiecewiseLinearFillPatch class:**



Linear interpolation of coarse-level data (time and space) into fine ghost cells.

**example:**

```
ProblemDomain crseDomain;
DisjointBoxLayout crseGrids, fineGrids;
int nComp, refRatio, nGhost;
Real oldCrseTime, newCrseTime, fineTime;

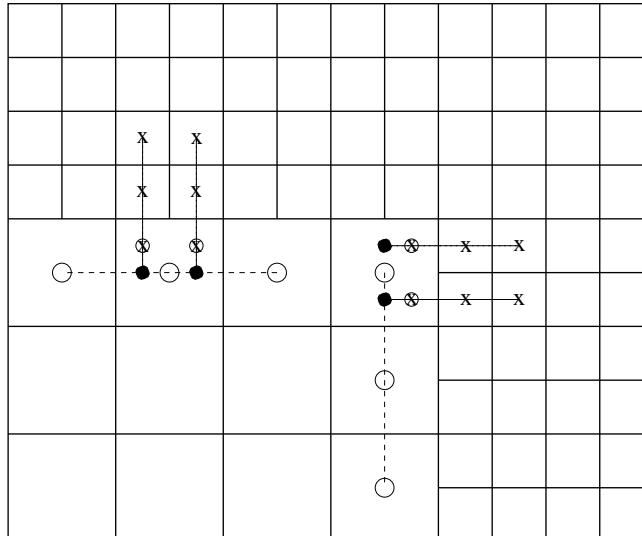
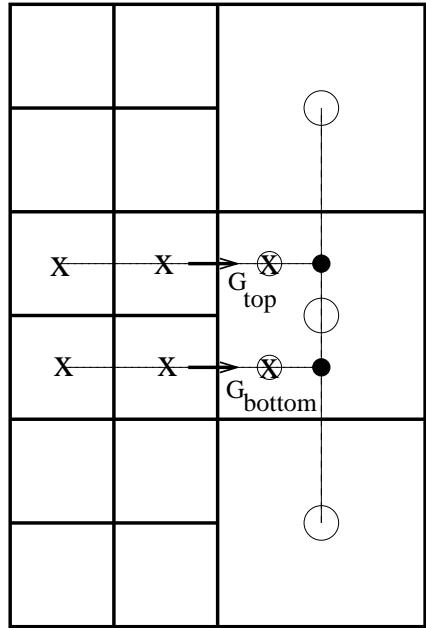
LevelData<FArrayBox> fineData(fineGrids, nComp, nGhost*IntVect::Unit)
LevelData<FArrayBox> oldCrseData(crseGrids, nComp);
LevelData<FArrayBox> newCrseData(crseGrids, nComp);

PiecewiseLinearFillPatch filler(fineGrids, coarseGrids, nComp,
                                crseDomain, refRatio, nGhost);

Real alpha = (fineTime-oldCrseTime)/(newCrseTime-oldCrseTime);

filler.fillInterp(fineData, oldCrseData, newCrseData, alpha,
                  0, 0, nComp);
```

**Example:** class QuadCFInterp



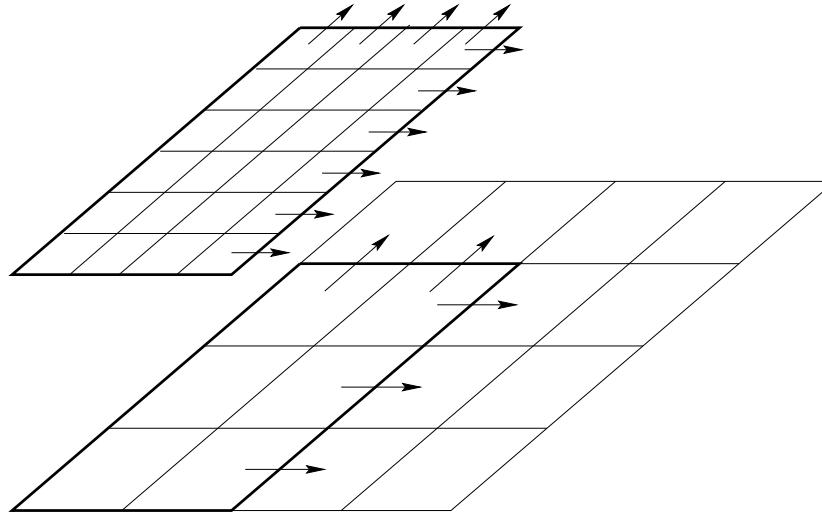
Many elliptic operator discretizations require quadratic interpolation involving the coarse- and fine-grid data.

A QuadCFInterp object encapsulates this functionality:

```
// define stencils, etc for the given grid configurations
QuadCFInterp(const DisjointBoxLayout& a_fineBoxes,
              const DisjointBoxLayout* a_coarBoxes,
              Real a_dxFine,
              int a_refRatio,
              int a_nComp,
              const ProblemDomain& a_domf);

// fill in ghost cells of phif with interpolated values
QuadCFInterp::coarseFineInterp(LevelData<FArrayBox>& a_phif,
                                LevelData<FArrayBox>& a_phic);
```

**LevelFluxRegister class:**



$$U^c := U^c + \Delta t^c (F_{\mathbf{i}^c - \frac{1}{2}\mathbf{e}}^{c,s} - \frac{1}{Z} \sum_{\mathbf{i}^f} F_{\mathbf{i}^f - \frac{1}{2}\mathbf{e}}^{f,s})$$

The coarse and fine fluxes are computed at different times in the program, and on different processors. We rewrite the process in the following steps:

$$\delta F = 0$$

$$\delta F := \delta F - \Delta t^c F^c$$

$$\delta F := \delta F + \Delta t^f < F^f >$$

$$U^c := U^c + D_R(\delta F)$$

A `LevelFluxRegister` object encapsulates these operations:

- `LevelFluxRegister::setToZero()`
- `LevelFluxRegister::incrementCoarse`: given a flux in a direction for one of the patches at the coarse level, increment the flux register for that direction.
- `LevelFluxRegister::incrementFine`: given a flux in a direction for one of the patches at the fine level, increment the flux register with the average of that flux onto the coarser level for that direction.
- `LevelFluxRegister::reflux`: given the data for the entire coarse level, increment the solution with the flux register data for all of the coordinate directions.

### **Layer 3: Reusing Control Structures Via Inheritance (`EllipticDesign`, `AMRTimeDependent`).**

AMR has multilevel control structures which are largely independent of the details of the operators and the data.

- Multigrid iteration on an AMR hierarchy. (multilevel AMR solve)
- Berger-Oliger timestepping (refinement in time).

To separate the control structure from the details of the operations that are being controlled, we use C++ inheritance in the form of *interface classes*.

## Elliptic Solver Example: LinearSolver virtual base class

```
class LinearSolver<T>
{
// define solver
virtual void define(LinearOp<T>* a_operator, bool a_homogeneous) = 0;

// Solve L(phi) = rhs
virtual void solve(T& a_phi, const T& a_rhs) = 0;
...
}
```

LinearOp<T> defines what it means to evaluate the operator (for example, a Poisson Operator) and other functions associated with that operator. T can be an FArrayBox (single-grid), LevelData<FArrayBox> (single-level), Vector<LevelData<FArrayBox>\*> > (Multilevel AMR hierarchy), etc

- applyOp(..)
- residual(..)

The use of *interface classes* such as this one is a common idiom in Layer 3 tools. It allows one to reuse control structures (multigrid iteration, Berger-Oliger time-stepping) by using inheritance to define the interface to the operator.

LinearOp-derived operator classes:

- **AMRPoissonOp** (constant-coefficient Poisson's equation on AMR hierarchy)  
 $(\alpha \vec{I} - \nabla \cdot \beta \nabla) \phi = \rho$
- **VCAMRPoissonOp** (variable coefficient version)

LinearSolver-derived control-structure classes:

- **AMRMultiGrid** – use multigrid to solve an elliptic equation on a multilevel hierarchy of grids, using composite AMR operators. If base level  $\ell_{base} \neq 0$ , uses interpolated boundary conditions from coarser level.
- **BiCGStabSolver** – Use BiConjugate Gradient Stabilized method to solve an elliptic equation (can be single-level, or multilevel). Useful for variable-coefficient problems with strongly-varying coefficients. Also useful as a “bottom solver” for AMRMultiGrid

## Factory Classes:

Instead of a single `linearOp`, `AMRMultiGrid` needs a set of `AMRPoissonOp`'s (one for each AMR level, along with auxiliary levels required by multigrid.)

Solution: Factory classes (`AMRPoissonOpFactory`)

- Factory class contains everything necessary to define the appropriate operator class at any required spatial resolution.
- Define function for AMR hierarchy:

```
void define(const ProblemDomain& a_coarseDomain,
            const Vector<DisjointBoxLayout>& a_grids,
            const Vector<int>& a_refRatios,
            const Real&           a_coarsedx,
            BCHolder a_bc,
            Real    a_alpha = 0.,
            Real    a_beta  = 1.);
```

- `AMRPoissonOp* AMRnewOp( const ProblemDomain& a_indexSpace )` function – returns an `AMRPoissonOp` defined for the given level.

```
int numlevels, baselevel;
Vector<DisjointBoxLayout> vectGrids;
Vector<ProblemDomain> vectDomain;
Vector<int> vectRefRatio;
Real dxCrse;
setGrids(vectGrids, vectDomain, dxCrse,
          vectRefRatio, numlevels, baselevel);

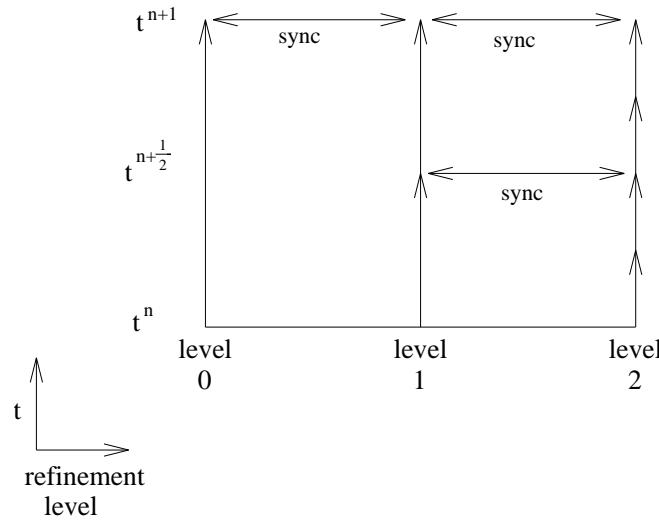
AMRPoissonOpFactory opFactory;
opFactory.define(vectDomain[0], vectGrids, vectRefRatio,
                 dxCrse, &bcFunc);

AMRMultiGrid solver;
solver.define(vectDomain[0], opFactory, &bottomSolver, numLevels);

Vector<LevelData<FArrayBox>*> phi(numlevels, NULL);
Vector<LevelData<FArrayBox>*> rhs(numlevels, NULL);
defineStorageAndRHS(phi, rhs, vectGrids);

solver.solveAMR(phi, rhs, numlevels-1, baseLevel);
```

## Example: AMR / AMRLevel interface for Berger-Oliger timestepping



We implement this control structure using a pair of classes.

class AMR: manages the Berger-Oliger time-stepping process.

class AMRLevel: collection of virtual functions called by an AMR object that perform the operations on the data at a level, e.g.:

- `virtual void AMRLevel::advance() = 0` advances the data at a level by one time step.
- `virtual void AMRLevel::postTimeStep() = 0` performs whatever synchronization operations required after all the finer levels have been updated.

AMR has as member data a collection of pointers to objects of type `AMRLevel`, one for each level of refinement:

```
Vector<AMRLevel*> m_amrlevels;
```

AMR calls the various member functions of `AMRLevel` as it advances the solution in time:

```
m_amrlevels[currentLevel]->advance( );
```

The user implements a class derived from `AMRLevel` that contains all of the functions in `AMRLevel`:

```
class AMRLevelUpwind : public AMRLevel
// Defines functions in the interface, as well as data.
...
virtual void AMRLevelUpwind::advance()
{
// Advances the solution for one time step.
...
}
```

To use the AMR class for this particular application, `m_amrlevel[k]` will point to objects in the derived class, e.g.,

```
AMRLevelUpwind* amrLevelUpwindPtr = new AMRLevelUpwind(...);
m_amrlevel[k] = static_cast <AMRLevel*> (amrLevelUpwindPtr);
```

## Upwind Advection Solver

- Simple constant-velocity advection equation:

$$\frac{\partial U}{\partial t} + \vec{A} \nabla \cdot U = 0$$

- Discretize on AMR grid using simple 1st-order upwind approach.  
Piecewise-linear interpolation in space for coarse-fine boundary conditions.
- Refinement in time: linear interpolation in time for coarse-fine boundary conditions, since  $U$  is a conserved quantity, maintain conservation at coarse-fine interface using refluxing.

## Using Chombo AMRTimeDependent library for Upwind advection Equation

- `AMRLevelUpwind`: public `AMRLevel` class – derived from base `AMRLevel` class, fills in the specific functionality necessary for implementing the upwind advection equation solver algorithm:
  - `advance()` – advance a single AMR level by one timestep using 1st-order upwind.
  - `postTimeStep()` – synchronization operations: flux correction, average finer levels onto covered regions.
  - `tagCells(IntVectSet& tags)` – specify which cells on a given AMR level will be refined.
  - `regrid(const Vector<Box>& newGrids)` – given a new grid configuration for this level, re-initialize data.
  - `initialData()` – initialize data at the start of the computation.
  - `computeDt()` – compute the maximum allowable timestep based on the solution on this level

- `AMRLevelUpwindFactory`: public `AMRLevelFactory` class – derived from base `AMRLevelFactory` class. Used by AMR to define a new `AMRLevelUpwind` object.
  - `virtual AMRLevel* new_amrlevel() const` – returns a pointer to a new `AMRLevelUpwind` object.
  - Can also use to pass information through to all `AMRLevelUpwind`'s in a consistent manner (ex. advection velocity, CFL number)

## Sample Main program

```
{  
    // Set up the AMRLevel... factory  
    AMRLevelUpwindFactory amrLevelFact;  
    amrLevelFact.CFL(cfl);  
    amrLevelFact.advectionVel(advection_velocity);  
  
    AMR amr;  
  
    // Set up the AMR object with AMRLevelUpwindFactory  
    amr.define(maxLevel,refRatios,probDomain,&amrLevelFact);  
  
    // initialize hierarchy of levels from scratch for AMR run  
    amr.setupForNewAMRRun();  
  
    amr.run(stopTime,nstop);  
  
    amr.conclude();  
}
```

```
// Advance by one timestep
Real AMRLevelUpwind::advance( )
{
    // Copy the new to the old
    m_UNew.copyTo(m_UOld);

    // fill in ghost cells, if necessary
    AMRLevelUpwind* coarserLevelPtr = NULL;
    // interpolate from coarser level, if appropriate
    if (m_level > 0)
    {
        coarserLevelPtr = getCoarserLevel();

        // get old and new coarse-level data
        LevelData<FArrayBox>& crseDataOld = coarserLevelPtr->m_UOld;
        LevelData<FArrayBox>& crseDataNew = coarserLevelPtr->m_UNew;
        const DisjointBoxLayout& crseGrids = crseDataNew.getBoxes();

        Real newCrseTime = coarserLevelPtr->m_time;
        Real oldCrseTime = newCrseTime - coarserLevelPtr->m_dt;
```

```
Real coeff = (m_time - oldCrseTime)/coarserLevelPtr->m_dt;

const ProblemDomain& crseDomain = coarserLevelPtr->m_problem_domain;
int nRefCrse = coarserLevelPtr->refRatio();
int nGhost = 1;

PiecewiseLinearFillPatch filpatcher(m_grids, crseGrids,
                                     m_UNew.nComp(), crseDomain,
                                     nRefCrse, nGhost);

filpatcher.fillInterp(m_UOld, crseDataOld, crseDataNew,
                      coeff, 0, 0, m_UNew.nComp());

}

// exchange copies overlapping ghost cells on this level
m_UOld.exchange();

// now go patch-by-patch, compute upwind flux, and do update
```

```
// iterator will only reference patches on this processor
for (dit.begin(); dit.ok(); ++dit)
{
    const Box gridView = m_grids.get(dit());
    FArrayBox& thisOldSoln = m_UOld[dit];
    FArrayBox& thisNewSoln = m_UNew[dit];
    FluxBox fluxes(gridView, thisOldSoln.nComp());

    // loop over directions
    for (int dir=0; dir<SpaceDim; dir++)
    {
        // note that gridbox will be the face-centered one
        Box faceBox = fluxes[dir].box();
        FORT_UPWIND(CHF_FRA(fluxes[dir]),
                    CHF_FRA(thisOldSoln),
                    CHF_REALVECT(m_advectionVel),
                    CHF_REAL(m_dt),
                    CHF_REAL(m_dx),
                    CHF_BOX(faceBox),
                    CHF_INT(dir));
    }
}
```



```
    } // end loop over directions

    // do flux difference to increment solution
    thisNewSoln.copy(thisOldSoln);

    for (int dir=0; dir<SpaceDim; dir++)
    {
        FORT_INCREMENTDIVDIR(CHF_FRA(thisNewSoln),
                             CHF_FRA(fluxes[dir]),
                             CHF_BOX(gridBox),
                             CHF_REAL(m_dx),
                             CHF_REAL(m_dt),
                             CHF_INT(dir));
    }
}

} // end loop over grid boxes

// Update the time and store the new timestep
m_time += m_dt;
return m_dt;
```



## Layer 4: AMR Applications

- A general driver for an unsplit second-order Godunov method for hyperbolic conservation laws. User provides physics-dependent components (characteristic analysis, Riemann solver).
- AMR multigrid solvers for Poisson, Helmholtz equations.
- Incompressible Navier-Stokes solver using projection method. Includes projection operators for single level, AMR hierarchy. Advection-diffusion solvers.
- Wave equation solver.
- Volume-of-fluid algorithm fluid-solid interactions.

## AMR Utility Layer

- BRMeshRefine –Given IntVectSets of “tagged” cells for refinement, creates Vectors of Boxes which define an AMR Hierarchy using the Berger-Rigoutsos algorithm. These Vectors of boxes can then be distributed (using some sort of load balancing estimation) and used to create Vectors of disjointBoxLayouts.
- API for HDF5 I/O.
- Interoperability tools. We are developing a framework-neutral representation for pointers to AMR data, using opaque handles. This will allow us to wrap Chombo classes with a C interface and call them from other AMR applications.
- Chombo Fortran - a macro package for writing dimension-independent Fortran and managing the Fortran / C interface.
- ParmParse class from BoxLib for handling input files.
- Visualization and analysis tools (ChomboVis, VisIt).

## **Creating a hierarchy of levels using `BRMeshRefine`**

Start with `IntVectSets` of “tagged” cells on each coarse level defining regions where refinement is desired.

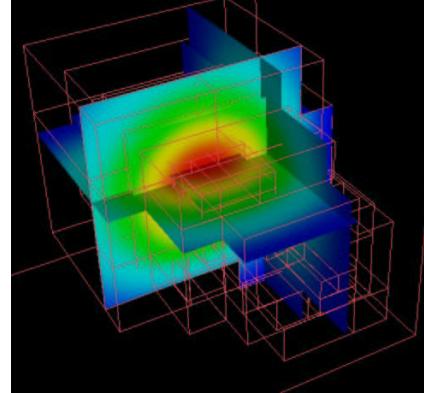
### **MeshRefine Parameters:**

- `fillRatio` – “efficiency of grids” = (number of refined cells)/(number of tagged cells)
- `blockFactor` – factor by which grids will be “coarsenable” (important for multigrid)
- `bufferSize` – “proper nesting” radius – minimum number of level  $\ell$  cells between  $(\ell + 1)/\ell$  and  $\ell/(\ell - 1)$  interfaces
- `maxSize` – longest allowable grid size in a single direction

### **example:**



## ChomboVis Interactive Visualization and Analysis Tools



- “AMR-aware”
  - Block-structured representation of the data leads to efficiency.
  - Useful as a debugging tool (callable from debuggers (gdb))
- Visualization tools based on VTK, a open-source visualization library.
- Implementation in C++ and Python
  - GUI interface for interactive visualization
  - Command-line python interface to visualization and analysis tools, batch processing capability – goal is a full analysis tool.
- Interface to HDF5 I/O along with C API provides access to broad range of AMR users. (“Framework-neutral”)

Chombo, ChomboVis available from the ANAG website:

- <http://seesar.lbl.gov/ANAG/software.html>
- Latest Release: May, 2007.

## **Acknowledgements**

DOE Applied Mathematical Sciences Program

DOE HPCC Program

DOE SciDAC Program

NASA Earth and Space Sciences Computational Technologies Program